Electronic Notes in Theoretical Computer Science 82 No. 7 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 14 pages

A Hierarchical Program Representation for Refactoring

Niels Van Eetvelde, Dirk Janssens [1,2]

Department of Mathematics and Computer Science
University of Antwerp
Wilrijk, Belgium

Abstract

Currently there is a lot of interest in graph representations of software systems, as they provide a natural and flexible means to describe complex structures. The various visual sublanguages of the UML are perhaps the most obvious example of this. In [11] a graph representation of object-oriented programs was presented that enables one to describe refactoring operations (behaviour-preserving changes in the structure of a program) in a formal, concise way by graph rewriting productions. In general, however, a refactoring makes changes to a small part of a program, so the graph representation should only contain the information needed to carry out that refactoring. All other details are redundant and make the graph unnecessarily large for good visualization. A possible solution consists in using a hierarchical representation. Such a representation of object-oriented programs is presented in this paper. It is based on node-rewriting graph productions: each refinement step corresponds to a production. The construction is illustrated by applying it to a small Java simulation of a Local Area Network.

1 Introduction

It has been widely recognized that graphs provide an interesting formalism for modeling the complex structures that occur in software engineering. The various visual sublanguages of the UML are perhaps the most well-known example of this [15].

One application of graph-based techniques was presented in [11]: the description of refactoring operations by graph transformation [14]. Refactorings

[1] Email: Niels.Vaneetvelde@ua.ac.be, Dirk.Janssens@ua.ac.be

[2] This research is funded by FWO research grant G.0452.03: "A formal foundation for software refactoring" and is supported by the EU TMR contract SegraVis through University of Antwerp - UIA.

[6,13] are operations that are used to improve the structure of a program without changing its behaviour. Most of the work on this topic has been done in the area of object - oriented programs. A list of refactoring operations can be found in [6].

The graph representation of [11] is rather detailed: it contains information about every class, and most of the control flow details for every method. Thus, even for relatively small programs, the graphs involved are large and not really useful for visualization. Moreover, we are not always interested in the finest program details, when studying e.g. the effects of a refactoring operation at class level. In this paper a possible solution to this problem is investigated: the introduction of a hierarchical structure for the graphs that represent programs. As an example of the ideas from [11], consider the graph rewriting production depicted in Figure 1. It represents the *Pull Up Method* refactoring, moving a method one level higher in the inheritance hierarchy. The nodes marked *m M*, *MD*, *Parent C* and *Child C* represent the method signature, the method definition, and the two classes involved. The edges with label *l* (lookup), *m* (member) and *i* (inheritance) represent the fact that *m M* is the signature of *MD*, *MD* is a member of *Child C* and *Child C* is a subclass of *Parent C*. A refactoring such as *Pull Up Method* may happen only in a context that satisfies certain requirements which cannot be specified directly in the left - hand side of the production, but which can be expressed as negative context conditions [7]. E.g., in the case of this particular refactoring the method body that is moved should not access variables that are defined in *Child C*, since these may have a different meaning in the context of *Parent C*. This can be presented by the forbidden context of Figure 2, where the *a* - labeled edge represents the fact that *MD* accesses the variable name *V* which is the name of a member *VD* of class *Child C*. Formally, *Pull Up Method* may be applied only to those occurrences of its left-hand side that cannot be extended by the graph of Figure 2 (by gluing the graphs together over nodes 2 and 3).

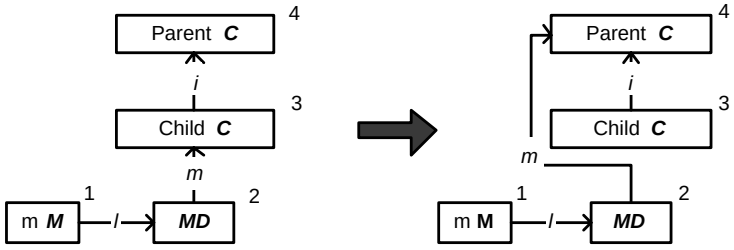

Fig. 1. The *Pull Up Method* refactoring

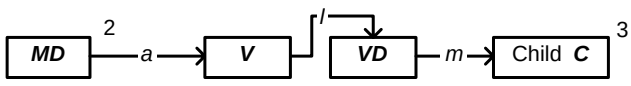

Fig. 2. A negative context condition for the refactoring

When describing realistic software, with perhaps hundreds of classes and methods, with multiple membership, inheritance and access relations between

them, the program graphs become very complex. A careful analysis of the *Pull Up Method* refactoring shows that the first step of the refactoring, recognizing the pattern on the left hand side, can be carried out using only knowledge about the l, m and i edges, whereas checking the context condition requires knowledge about the fact that *MD* accesses *V* (which is represented by a - edges in the program graph). It would be desirable to treat the first step (picking a candidate occurrence) using a graph representation where the access information is hidden, then refine (zoom in) the graph to check the negative precondition, then undo the refinement step again (zoom out) and finally execute the transformation. In this way the various phases of a refactoring operation (finding a candidate occurrence, checking context conditions, transforming the program structure) can each be treated at the appropriate level of abstraction, using program graphs in which only the relevant parts are refined. The parts of the graph that are not required for the particular operation or check that one is interested in, may remain at a higher, less detailed level.

It is demonstrated in Section 3 that graph rewriting can be used to achieve this: four levels of abstraction are defined, and it is shown that the necessary refinements correspond to graph productions. Each of these productions rewrites just one node, and thus the derivation of a graph may be viewed as a tree. This implies that the partially refined graphs may be viewed as hierarchical graphs [4,3]. In Section 4 it is discussed how the *Pull Up Method* and the *Inline Method* refactorings behave with respect to the four levels. These levels are chosen to fit the intended application (i.e. visualizing refactoring operations), but at the same time they seem natural from the viewpoint of program design, where classes are first refined by determining their members, then the members (in particular, methods) are refined by specifying the way they access variables or methods and finally the methods are refined into concrete implementations (i.e. executable code). In spite of this fact, it seems possible to use a similar methodology (refinement by node rewriting) for other kinds of refinement in which other information (e.g. pertaining to real-time behaviour or instrumentation) is added stepwise to the elements of program graphs.

2 Running Example

Throughout the paper, a small Java program is used as a running example: it is a program simulating a simple Local Area Network [2]. It consists of four classes: the `Node` and `Packet` classes embody the base system of network nodes sending packets to each other. The `Workstation` and `PrintServer` classes are used for nodes that support additional services, like printing or creating packets. For the sake of the presentation, only the essence of the code is shown here.

```
public class Node {
  public String name;
```

```
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p);
  }
  protected void send(Packet p) {
    System.out.println(name + nextNode.name);
    this.nextNode.accept(p);
  }
}
public class Packet {
  public String contents;
  public Node originator;
  public Node addressee;
}
public class PrintServer extends Node {
  public void print(Packet p) {
    System.out.println(p.contents);
  }
  public void accept(Packet p) {
    if(p.addressee == this) this.print(p);
    else super.accept(p);
  }
}
public class Workstation extends Node {
  public void originate(Packet p) {
    p.originator = this;
    this.send(p);
  }
  public void accept(Packet p) {
    if(p.originator == this)
      System.err.println("no destination");
    else super.accept(p);
  }
}
```

3 The hierarchical model

This section contains the main contribution of the paper: a hierarchical representation of object-oriented programs, based on graph rewriting. Four levels of abstraction are defined, and the refinements enabling one to pass from a higher level of abstraction to a lower one are described by graph productions. The productions used are based on node replacement and embedding, similar to those of [8] and [9]. The left-hand sides of the productions are one-node graphs, and hence each derivation can be represented by a tree; this tree defines a hierarchical structure on the graph produced.

In order for this approach to work, it is essential that each derivation tree

determines a unique graph, i.e. that the relative order of the refinement steps is irrelevant for the graph that results from them. The problem is illustrated in Figure 3: refinements of nodes x an y yield subgraphs g_x and g_y in the refined graph, which is represented by the dashed line. However, in general there are edges between nodes of g_x and nodes of g_y, and these should not depend on the question whether x is refined before y or vice versa. In the terminology of graph rewriting, it is needed that the set of productions is *confluent*. In [8] and [9] confluency is considered for a class of node - rewriting graph rewriting systems that contains the system considered in this paper. The relabelings of edge labels in the embedding are viewed as an algebraic operation, and it is shown that the system is confluent if those operations commute. To show that the system considered in this paper is confluent, however, an easier, more specialized reasoning will be given in Section 3.5. It has also been shown [10] that these graph productions correspond to sets of graph productions in the more common double-pushout approach to graph rewriting [1]. However, the resulting DPO productions may have left-hand sides that contain more than one node, so that they do not give rise to derivation trees in an obvious way.

Note that the fact that the tree uniquely determines the derived structure is trivial for syntax trees in the usual sense: these trees are ordered, and each leaf represents a symbol. The derived structure (a program) is the yield of the tree. In the case of node rewriting graph grammars, the trees are not ordered, and the leaves represent only the nodes, whereas the edges are either part of the right-hand sides of production occurrences, or produced by the embedding mechanism. As a result, the relative order of derivation steps is generally required to determine the derived graph and thus confluency is not a trivial property. Also, note that the hierarchical structure of program graphs introduced here is evidently somewhat similar to the syntactic structure of the corresponding program, but it is also different: the use of a particular graph representation implies a decision about which entities are represented by nodes, and which ones by edges. Nodes and edges play quite different roles in the graph rewriting process.

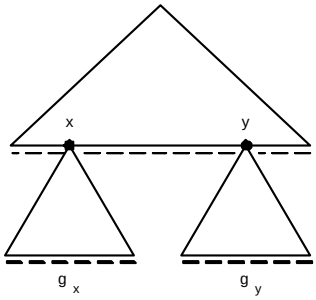

Fig. 3. Confluency is not trivial when there are edges between g_x and g_y

3.1 Level 1: Names and types

At the highest level of abstraction, only the names (variables and signatures) and types (classes) are represented. Thus there are three kinds of nodes, which can be distinguished by their labels: a node with label $n.M$ represents a method signature with name n and, similarly, labels $n.V$ and $n.C$ are used for nodes representing variable names and class names. One $n.M$ node is provided for every subtree of the inheritance hierarchy where that signature is known. The edges also carry labels, indicating which relationship holds between the source and the target. The edge labels present at this level are given in Table 1. The representation of the LAN simulation program is given in Figure 4.

Edge	Description
$(n.M \text{ or } n.V) \xrightarrow{t} m.C$	Name n is of type m
$(n.M \text{ or } n.V) \xrightarrow{n.d} m.C$	Name n is defined in class m
$n.M \xrightarrow{i.p} m.C$	The i'th parameter of n is of type m
$n.C \xrightarrow{i} m.C$	Class n extends (directly) class m
$n.C \xrightarrow{m.v} (m.M \text{ or } m.V)$	Name m is visible to class n

Table 1
The edge labels and the kinds of nodes involved

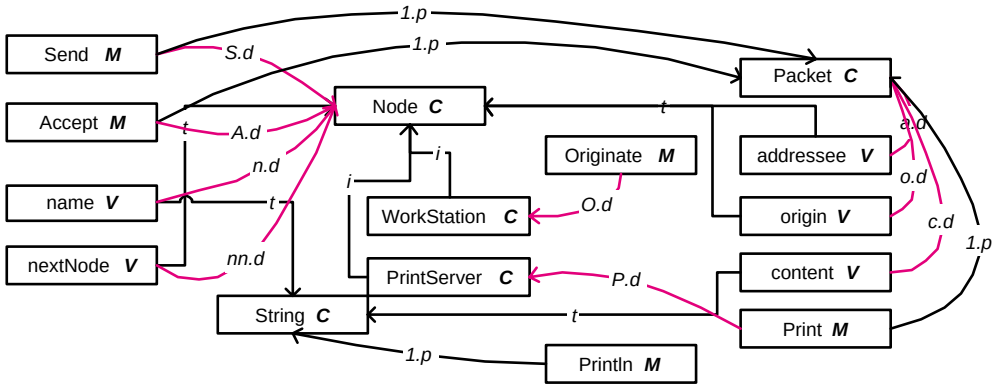

Fig. 4. The Lan simulation at level 1

In spite of the fact that most details of the program are hidden at this level, a number of edges are omitted in Figure 4: since in this example there are no constructs restricting the visibility of names (e.g. Java packages), all public names are visible to all classes. This will not be true anymore if one describes a more complete version of Java, including constructs restricting the visibility of names (such as packages or private names). For example, there is an *origin.v* edge from the *Node.C* node to the *origin.V* node, indicating that every method implementation in the `Node` class *may* use the `origin` public variable. Also, a name known to a superclass is also known to its

subclasses, and thus many of the edges with label $n.d$ can be inferred from the inheritance relationship. Only the $n.d$ edges that cannot be inferred in this way are represented, by dashed arrows. E.g., `Originate` is only defined in `Workstation` but not in its parent `Node`. Name, however, is defined on classes `Node`, `Workstation` and `PrintServer` The v and d edges are needed for the embedding mechanism of the refinement steps, the reason they carry the same name as the variable or signature node they are connected to will become clear in the next sections.

3.2 Level 2: Membership

At this level it is specified where each variable and method is implemented. Thus at this level for each class a complete class definition (without an implementation) is given (see Figure 5). Formally, two additional node labels and two additional edge labels are introduced: the method bodies and variables are represented by nodes with label *MD* and *VD*, and the new edge labels and their interpretations are given in Table 2.

Edge	Description
$n.M \xrightarrow{l} MD$	Method n has an implementation *MD*
$n.V \xrightarrow{l} VD$	Variable name n has a definition *VD*
$VD \xrightarrow{m} n.C$	*VD* represents a variable definition in class n
$MD \xrightarrow{m} n.C$	*MD* represents a method definition in class n
$MD \xrightarrow{n.v} (n.M \text{ or } n.V)$	Name n is visible to the method implementation *MD*

Table 2
Edges added at level 2

The step from level 1 to level 2, refining a class node, is described by the production of Figure 5. The production is equipped with an *embedding relation*, given in Table 3. Its meaning is the following: two relations are given, one for incoming edges and one for outgoing edges of the rewritten node. Each of these relations specifies how the edges should be redirected and/or relabeled, becoming edges incident to the nodes of the right-hand side of the production. E.g., the first line of the "Incoming edges" relation means that incoming edges with label *Send.d* become incoming edges of node 2 with label l. Part of the LAN program after rewriting the `Node` and `PrintServer` class nodes is depicted in Figure 6. Edges with label $n.d$ or $n.v$ are omitted.

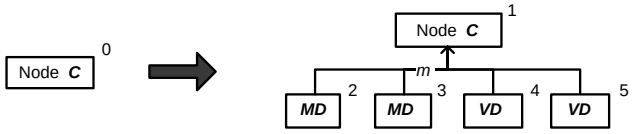

Fig. 5. Production introducing members of a class

Incoming edges	Outgoing edges
$(Send.d, 0) \to (l, 2)$	$(i, 0) \to (i, 1)$
$(Accept.d, 0) \to (l, 3)$	for every name n:
$(name.d, 0) \to (l, 4)$	$(n.v, 0) \to (n.v, 2), (n.v, 3)$
$(nextnode.d, 0) \to (l, 5)$	
for every $x \in \{i, i.p, t\}$: $(x, 0) \to (x, 1)$	

Table 3
The embedding relations

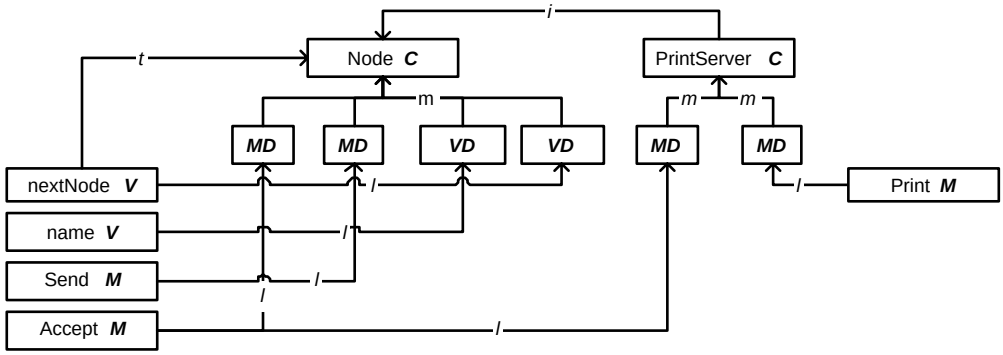

Fig. 6. Part of the example program at level 2

3.3 Level 3: Access/Update/Call

Until now, the representation contains no information about the way names are used in classes or methods. The next level refines the visibility relationship, providing this information: we distinguish between calls to methods, updates of variables and other (read-)access to variables. No new nodes are added to the representation, but edges with label $n.v$ are replaced by edges with label c, u and a connecting the method bodies to the names that are called, updated and accessed. The refinement can again be described by a graph production; for the implementation of the `Send` method of `Node` the production is given in Figure 7 and Table 4. Figure 8 shows the changed part of the LAN graph.

Fig. 7. Production refining the visibility relation

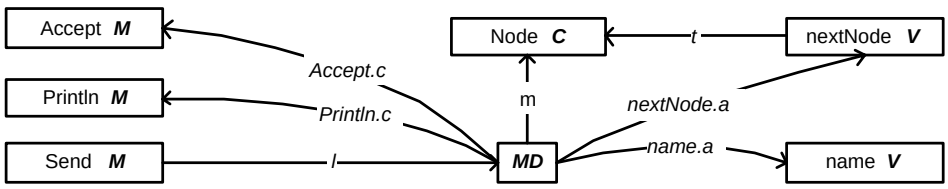

Fig. 8. Part of the LAN simulation at level 3

Incoming Edges	Outgoing edges
$(l,0) \to (l,1)$	$(m,0) \to (m,1)$ $(Accept.v,0) \to (Accept.c,1)$ $(Println.v,0) \to (Println.c,1)$ $(nextnode.v,0) \to (nextnode.a,1)$ $(name.v,0) \to (name.a,1)$

Table 4
The embedding relations

3.4 Level 4: Control flow

The last and most detailed level contains all the information from level 3, but also contains the full syntax tree representation of each method implementation. The edges labeled c, u and a are connected to the appropriate nodes of the syntax trees. Thus there is, e.g., an edge labeled u from a node nd of a syntax tree into a node with label $n.V$ if nd represents an update of variable n. Again, this refinement corresponds to a set of graph productions; its general form is given in Figure 9 , where the triangle represents a syntax tree of the usual form, and in Table 5. Figure 10 depicts part of the example program at level 4, after refining the `Send` method of `Node`.

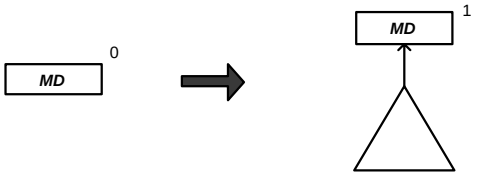

Fig. 9. Production refining a method definition

Incoming Edges	Outgoing edges
$(l,0) \to (l,1)$	$(m,0) \to (m,1)$ for each nd that represents an access to variable n: $(n.a,0) \to (n.a,nd)$ for each nd that represents an update to variable n: $(n.u,0) \to (n.u,nd)$ for each nd that represents a call to method n: $(n.c,0) \to (n.c,nd)$

Table 5
The embedding relations

3.5 Confluency

It still remains to show that the system presented is confluent. To this aim, it suffices to consider the situation of Figure 11, where productions p_1 and p_2 are used to rewrite the adjacent nodes x and y. For each x', y' in the right-hand

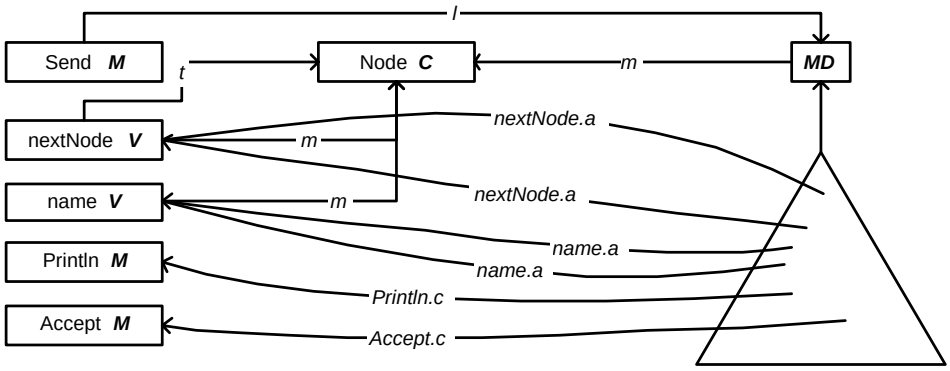

Fig. 10. Part of the LAN simulation at level 4

sides of p_1 and p_2, The embedding mechanism of p_1 and p_2 either transforms the edge from x to y into an edge from x' to y' or deletes it. The system is confluent if, in each such situation, the presence of an edge from x' to y', and its label a' does not depend on the order in which p_1 and p_2 are applied. For the system considered in this paper, this property holds: only nodes of type C and MD are rewritten. Two nodes of type MD are never adjacent, and if a node x of type MD is adjacent to a node y of type C, then x represents a member of y, and hence y cannot be rewritten after x is created. Thus the only possibility left is that both p_1 and p_2 of Figure 11 are nodes of type C and a is the label i. As described in Table 3, an i - labeled edge is left unchanged (redirected from node 0 to node 1 in Figure 5, without changing the label) when applying a production to a node of type C. It follows that also in this case the edge from x' to y' is not dependent on the order.

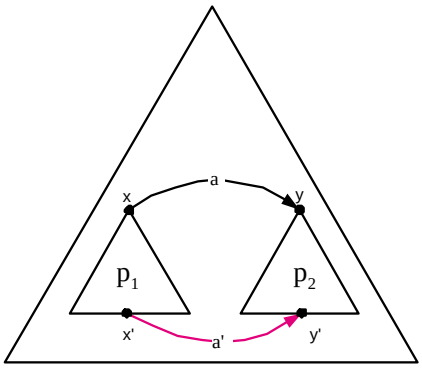

Fig. 11. Rewriting two connected nodes

4 Applications

The *Pull Up Method* refactoring discussed in the introduction fits nicely into this hierarchical system: the various phases (recognition of the left-hand side, checking of context conditions, transformation of the graph) require information at levels 2, 3 and 2 respectively.

The recognition of the left hand side of the graph pattern requires the model to be fully expanded to level 2, where all the edges with labels l, i and m are available. When an occurrence is detected using the level 2 representation, a local expansion to level 3 suffices for checking the condition of Figure 2: it suffices to rewrite node 2 of the right hand side (Figure 1) using the production of Figure 7. This illustrates the fact that the graphs used do not have to be uniform, in the sense that some parts are expanded to a greater level of detail than others, as illustrated in Figure 12.

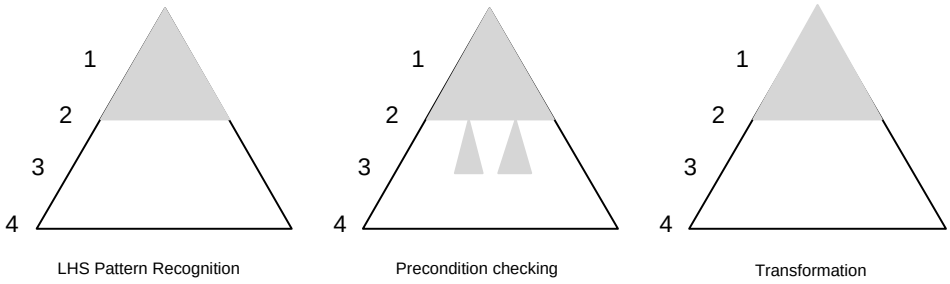

Fig. 12. The level of detail required for each of the refactoring phases of *Pull Up Method*

As another example, consider the *Inline Method* refactoring [6]. This operation replaces a function call with a copy of this function. This is frequently done by compilers to optimize code and it is sometimes useful to eliminate short methods and make code less complex. The refactoring is described schematically by the productions P_1 and P_2 depicted in Figures 13 and 14 and the negative context condition in Figure 15.

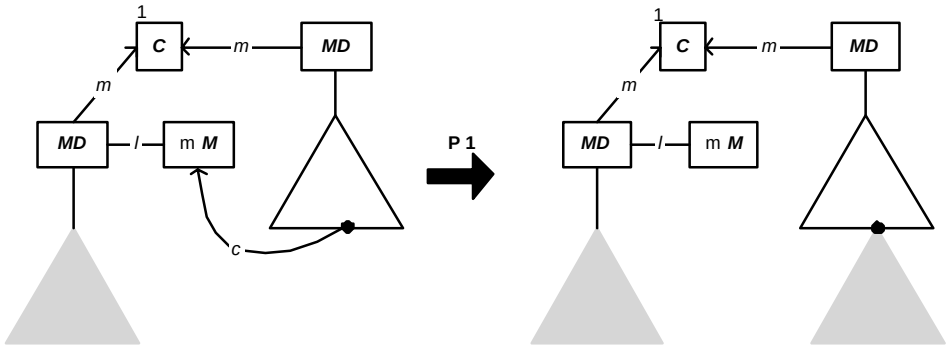

Fig. 13. Production replacing a call to m by the implementation of m

P_1 should be applied as long possible (replacing all calls to method m), and then P_2 should be applied once (removing m). The negative precondition states that the method to be inlined should not be called from another class than the class m belongs to. This implies that the refactoring is restricted to a single class and requires only information about this class and the methods defined in it. The method definition to be inlined, as well as the method definitions that contain a call to it, should be expanded to level 4. The question

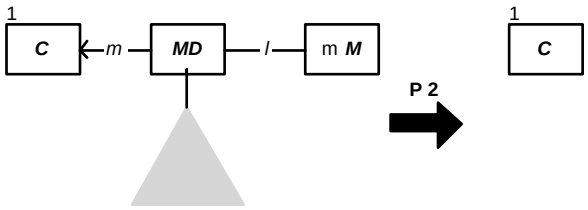

Fig. 14. Production removing method m

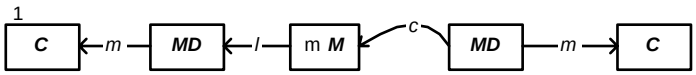

Fig. 15. Negative precondition for *Inline method*

whether a method definition contains such a call can be determined at level 3. Again each of the phases of the refactoring require different parts of the tree to be expanded, as sketched in Figure 16.

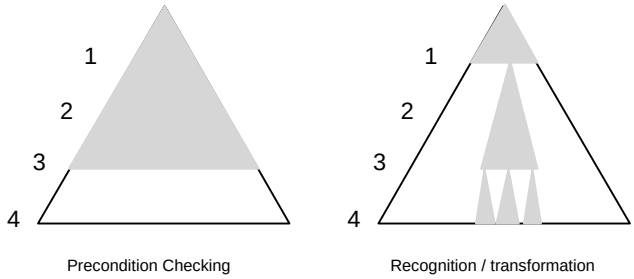

Fig. 16. The parts of the derivation tree needed for the *Inline Method* refactoring

5 Conclusion and future work

It has been shown that it is possible to define a hierarchical representation of object-oriented programs based on node-rewriting graph productions: each production corresponds to a refinement step. The set of productions involved is confluent, so that each derivation tree determines a graph representation of the program: the more derivation steps, the lower the abstraction level of the representation. The representation allows one to describe the various phases of refactoring at a suitable level of abstraction.

The list of refactoring operations in [6] is quite long and diverse. In order to obtain a suitable representation for each of them, we need to extend the proposed graph representation allowing one to represent program features and relations that have not been considered in this paper. Also, more powerful graph rewriting techniques are needed to specify program transformations concerning the copying or moving of (parts of) syntax trees. Such an extension would be necessary to express the *Move Method* or *Inline Method* refactoring.

Another issue that deserves more attention is the interaction between re-

finements and refactorings. It should be possible for a tool to detect automatically which refinements are needed to visualize or execute a particular refactoring. Other aspects are the possibility for a user to choose among various refinements, depending on what information he wants to see. Interesting topics are also the relationship to research about evolving hierarchical structures, and the treatment of other than object oriented languages.

In order to implement the concepts in this paper, one of the existing state of the art graph rewriting tools such as AGG, PROGRES and Fujaba will be used. We believe that the Fujaba tool[12] fits best our needs because of its tight integration with Java and the UML. Furthermore, the intuitive story diagrams[5] and extensible plugin architecture seem ideal to implement our refactoring ideas.

Acknowledgement

We like to thank Tom Mens and Bart Du Bois for their useful comments during the preparation of this paper.

References

[1] Corradini, A., H. Ehrig, R. Heckel, M. Löwe, U. Montanari and F. Rossi, *Algebraic approaches to graph transformation*, Handbook of Graph Grammars and Computing by Graph Transformation (1997), pp. 163–245.

[2] Demeyer, S., D. Janssens and T. Mens, *Simulation of a LAN*, Electronic Notes in Theoretical Computer Science **72** (2002).

[3] Drewes, F., B. Hoffmann and D. Plump, *Hierarchical graph transformation*, Lecture Notes in Computer Science **1784** (2000), pp. 98–113.

[4] Engels, G. and A. Schuerr, *Encapsulated hierachical graphs, graph types, and meta types*, in: *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, 1995.

[5] Fischer, T., J. Niere, L. Torunski and A. Zundorf, *Story diagrams: A new graph rewrite language based on the unified modeling language and java*, in: *TAGT*, 1998, pp. 296–309.

[6] Fowler, M., "Refactoring - Improving the Design of Existing Code," Addison Wesley, 1999.
URL `http://www.refactoring.com`

[7] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1996), pp. 287–313.

[8] Janssens, D., *ESM systems and the composition of their computations*, Graph Transformations in Computer Science **776** (1994), pp. 203–217.

[9] Janssens, D., "Handbook of Graph Grammars and Computing by Graph Transformation," World Scientific, 1999 pp. 56–106.

[10] Janssens, D., *Local action systems and DPO graph transformation*, Formal and Natural Computing **2300** (2002), pp. 138–157.

[11] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: *Graph Transformation*, Lecture Notes in Computer Science **2505** (2002), pp. 286–301, proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.

[12] Niere, J. and A. Zundorf, *Using fujaba for the development of production control systems*, Lecture Notes in Computer Science **1779** (2000), pp. 181–191.

[13] Opdyke, W. F., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA (1992).

[14] Rozenberg, G., "Handbook of Graph grammars and Computing by Graph Transformation," World Scientific, 1997.

[15] Rumbaugh, J., I. Jacobson and G. Booch, "The Unified Modelling Language Reference Manual," Addison - Wesley, 1998.